Cap 2: CPU

Intel 8086

El CPU se puede dividir de la siguiente manera…

1. **Datapath:**

Registro

Espacios donde almacenas la data en binario. Suelen tener tamanios de 16, 32 y 64 bits. Contienen datos y direcciones de memoria

ALU

Es la encargada de las operaciones lógicas y aritméticas (suma o multiplicación) que se requieren en la ejecución de un programa.

Dentro de ella esta conformada por compuertas logicas, mux, adders

2. **Unidad de Control**

Es la unidad encargada de "manejar el tráfico" del CPU monitoreando todas las instrucciones y la transferencia de información.

Arquitectura CISC (Complex Instruction Set Computer)

El ISA (Instruction Set Architecture) es lo que determina que una arquitectura sea CISC. Es como una lista de instrucciones que nos dice que cosas puede realizar (sea multiplicar, sumar, dividir, etc). La particularidad de este set de instrucciones es que hay un gran numero de instrucciones.

En una arquitectura CISC accedes directamente a memoria.

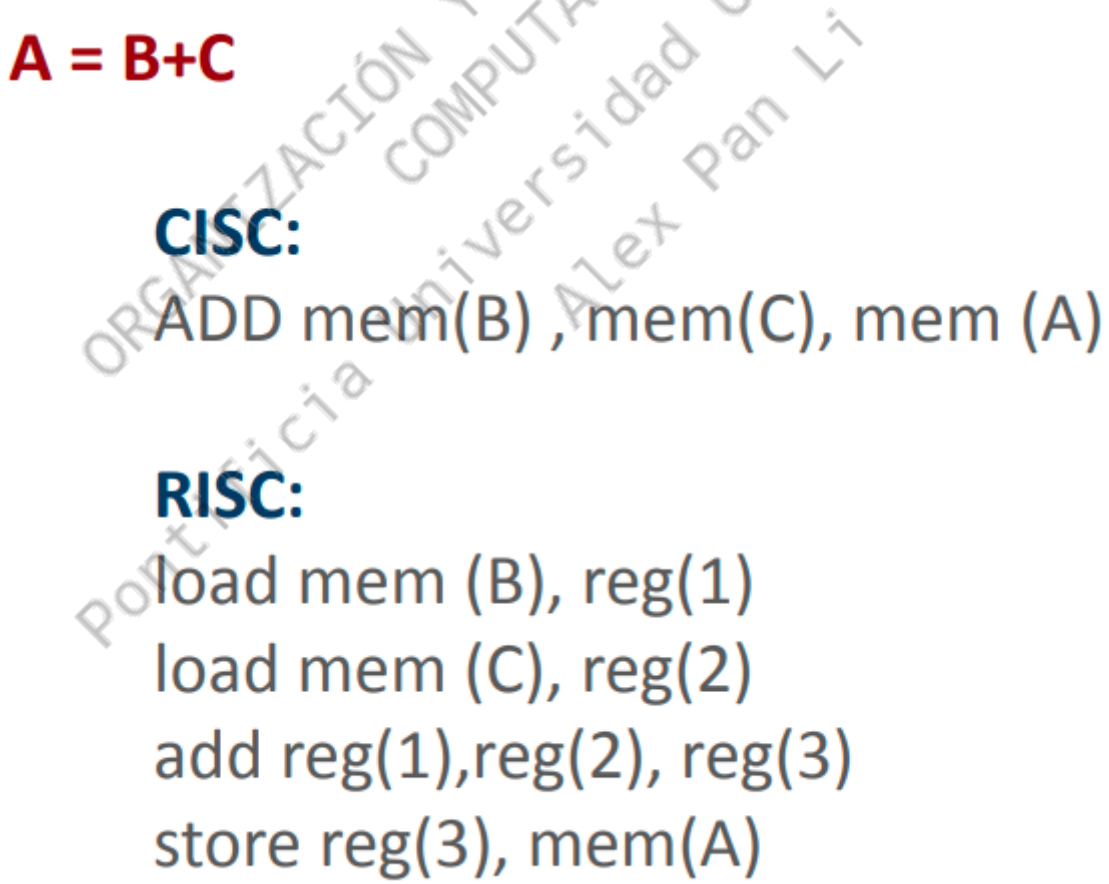
Arquitectura RISC (Reduced Instruction Set Computer)

Busca reducir el numero de instrucciones y su real objetivo es simplificar las instrucciones de tal forma que el programa se pueda ejecutar mas rapido. Tal es asi que solamente instrucciones como **LOAD** y **STORE** pueden acceder a memoria.

Asimismo, en esta arquitectura, todas las operaciones aritmeticas deben realizarse entre registros.

RISC	CISC
Multiples set de registros (más de 256)	Simple set de registro (6 a 16)
1 ciclo por instrucción (excepto load y store)	Múltiples ciclos por instrucción
Unidad de control cableado	Unidad de control micro-programada
Pocas instrucciones simples	Muchas instrucciones complejas
Pocos modos de direccionamiento	Muchos modos de direccionamiento
Solo load y store acceden a memoria	Muchas instrucciones acceden a memoria

Tengamos el siguiente ejemplo:



En una arquitectura CISC, una suma seria simplemente **ADD** + **posicion_de_memoriaB** + **posicion_de_memoriaC** + **posicion_de_memoriaA,** donde el ultimo argumento es la salida de la suma de input 1 con input 2

Sin embargo, en una arquitectura RISC, como solamente las instrucciones **LOAD** y **STORE** acceden a memoria, y las funciones aritmeticas solo funcionan entre registros, entonces mandamos lo que esta en B al registro 1, lo que esta en C al registro 2 y recien como ya estan en registros, podre sumar y guardarlo en un tercer registro 3. Finalmente, de ese resultado recien podre guardarlo en memoria A

Representancion Interna: Little Endian - Big Endian

El término “Endian” se refiere al orden de los bytes.

Es posible definir Little Endian y Big Endian a partir de un número entero que sea de dos bytes:

- **En representación Little endian:** El byte menos significativo primero seguido del byte más significativo. Por lo tanto, el byte en una dirección inferior tiene menor significado.

- **En representación Big endian:** El byte más significativo primero seguido del byte menos significativo. Por lo tanto, el byte en una dirección inferior tiene mayor significado.

Ejemplo: Se tiene una computadora en donde cada dirección guarda 1 byte y maneja números enteros de 32 bits. Mostrar en una tabla cómo se guardarán los números **0xABCD1234**, **0x00FE4321** y **0x10** en representación Big Endian y Little Endian. Asumir que se empezará desde la dirección 0x200.

Dirección	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

Arquitectura Interna de Procesadores Intel x86 (32 bits)

Como se puede observar, la arquitectura de 32 bits preserva los registros multiproposito general de la arquitectura de 16 bits (AX, BX, CX, DX) y extiende la capacidad de estos colocando la letra “E” a todos los registros (E de extended). De esta manera, de la arquitectura de 32 bits usamos EAX, y este nos permite usar AX y este nos permite usar AH y AL. Sin embargo, si decimos que EAX = 5, entonces AX = 5 y AL = 5 (realizar la asignacion EAX = 5 hace que AX y AL tambien valgan 5… si interactuamos AH, entonces tambien cambia AX y EAX). Tener en consideracion que podemos estar modificando la parte low de EAX pero no la parte high de EAX.

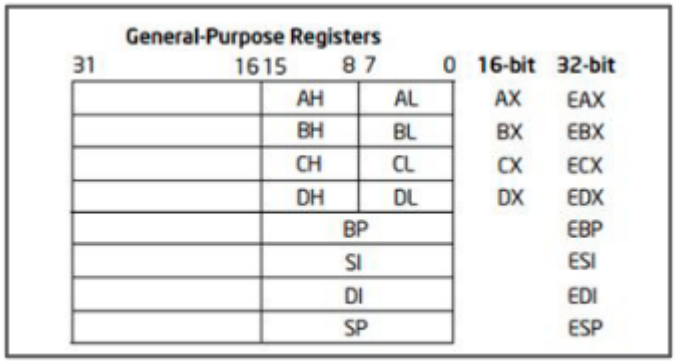


Figura 4. Registros de propósito general [3].

Por que Aprender Lenguaje Ensamblador?

Ventajas	Desventajas
Propósitos educacionales	Tiempo de desarrollo
Depuración y verificación	Depuración y verificación
Realizar compiladores	Fiabilidad y seguridad
Instrucciones no accesibles por alto nivel	Compiladores han mejorado en los últimos años
Optimización por tamaño y velocidad	Códigos de sistemas usan funciones intrínsecas (C++)

Ventajas

1. Es pedagogico: te ayuda a entender a un nivel mas detallado como funciona una computadora y como funciona un sistema operativo
2. En lenguaje ensamblador es mucho mas facil debuggear, ya que estas en el minimo detalle de la computadora.
3. Instrucciones no accesibles por alto nivel: Cuando codificamos en C, no tenemos tanto control, en cambio, en lenguaje ensamblador, somos capaces de optimizar funciones que para el compilador quizas no lo vea como una optimizacion.

Desventajas

1. Codificar en lenguaje ensamblador demora y debes aprenderte el ISA

2. La depuracion y verificacion esta ligado a la fiabilidad y seguridad… Esto se debe a que es mas complejo y si hacemos algo mal, entonces nos puede causar problemas (quemar circuitos inclusive) y podriamos incluso entrar a espacios que no deberiamos entrar

Generalmente, cuando se escribe alguna aplicación esta se subdivide en 3 secciones:

- **Sección data:** Declaración y definición de data inicial. Se declaran db, dw, dd y dq para 8, 16, 32 y 64 bits.
  seccion .data
- **Sección bss:** Declaración y definición de data sin valor inicial. Se declaran espacios como resb,resw,resd,resq.
  seccion .bss
- **Sección text:** Código de operación
  seccion .text

Este código se complementa con las interrupciones propias del sistema.

Programacion en Lenguaje Ensamblador: Sintaxis Intel y AT&T

Dependiendo de la arquitectura que se utilice, es posible fijar la sintaxis que se utilizará. Históricamente, para la programación de ensamblador en sistemas de cómputo se han utilizado las sintaxis de Intel y de AT&T.

Ejemplo Intel: MOV EAX,4

Ejemplo AT&T: MOVL $4, %EAX

	Intel		AT&T
1	section .data	1	.section .data
2	var1 dd 40	2	var1: .int 40
3	var2 dd 20	3	var2: .int 20
4	var3 dd 30	4	var3: .int 30
5	section .text	5	.section .text
6	global _start	6	.globl _start
7	_start:	7	_start:
8	mov ecx, [var1]	8	movl (var1), %ecx
9	cmp ecx, [var2]	9	cmpl (var2), %ecx
10	jg check_third_var	10	jg check_third_var
11	mov ecx, [var2]	11	movl (var2), %ecx
12	check_third_var:	12	check_third_var:
13	cmp ecx, [var3]	13	cmpl (var3), %ecx
14	jg _exit	14	jg _exit
15	mov ecx, [var3]	15	movl (var3), %ecx
16	_exit:	16	_exit:
17	mov eax, 1	17	movl $1, %eax
18	mov ebx, ecx	18	movl %ecx, %ebx
19	int 80h	19	int $0x80

Con estas primeras lineas, queremos asignar a la RAM los numeros 40, 20 y 30

```
var1 dd 40
var2 dd 20
var3 dd 30
```

Primero distingamos lo siguiente…

1. MOV EAX, 4 → EAX = 4
2. MOV ECX, [var1] → Voy a ir a var1 y voy a sacar el valor dentro, cosa que ECX = 4 (valor)
3. MOV ECX, var1 → Quiero que ECX tome el valor de lo que representa la etiqueta var1, osea que ECX = 0x1234
4. MOV ECX, var2 → Quiero que ECX tome el valor de lo que representa la etiqueta var2, osea ECX = 0x1238

Por lo tanto, en la linea 8, asigna el valor de 40 a ECX

```
mov ecx, [var1]
```

En la linea 9 compara lo que hay en ECX (que en este momento vale 40) con 20

```
cmp ecx, [var2]
```

Evaluamos si ECX > 20? → 40 > 20? VERDADERO, por lo tanto, saltamos a **check_third_var**

```
jg check_third_var
```

En la linea 13 comparamos lo que hay en ECX (que en este momento vale 40) con 30

```
cmp ecx, [var3]
```

Evaluamos si ECX > 30? → 40 > 30? VERDADERO, por lo tanto, saltamos a **_exit**

```
jg _exit
```

Lo que esta en **_exit** simplemente es para salir del programa

```
mov eax, 1
mov ebx, ecx
int 80h
```

Toolchain

- Es la cadena completa de programas que se usan para convertir el código fuente, en código de máquina, vincular entre sí los módulos de código ensamblados/compilados, desensamblar los binarios y convertir sus formatos.
- Inicializa su composición de la siguiente forma:

a. El **ensamblador** convierte los programas en lenguaje ensamblador en código binario. Esto lo hace generando los archivos de objeto (extensión .o).

b. El **enlazador (linker)** combina varios archivos objeto resolviendo sus referencias de símbolos externos y reubicando sus secciones de datos, generando un único archivo ejecutable.

- Si se utiliza sintaxis Intel, se debe utilizar el ensamblador **NASM**.
- Si se utiliza sintaxis AT&T, se debe utilizar el ensamblador **GAS.**

- Independientemente al ensamblador, el enlazador que se utilizará es **ld.**
- Si los objetos se deben enlazar con otros lenguajes, dependerá del tipo de lenguaje que se use para compilar. Por ejemplo, gcc también funciona como linkeador para objetos de ASM y códigos en C.

Arquitectura de Procesadores Intel x86

	64 Bits	32 Bits	16 Bits	8 Bits
Acumulador	RAX	EAX	AX	AH - AL
Base	RBX	EBX	BX	BH - BL
Contador	RCX	ECX	CX	CH - CL
Datos	RDX	EDX	DX	DH - DL
Source Index	RSI	ESI	SI	SIH - SIL
Destination Index	RDI	EDI	DI	DIH - DIL
Base Pointer	RBP	EBP	BP	BPH - BPL
Stack Pointer	RSP	ESP	SP	BPH - SPL
Propósito general	R8-R15	R8D-R15D	D8W	D8B

Bandera	Descripción
CF	Acarreo
PF	Paridad
ZF	Zero
SF	Signo
OF	Desborde
AF	Ajuste
IF	Interrupciones

Algo que se rescata de la arquitectura x86 es **SIMD** (Single Instruction Multiple Data), es decir, a traves de una instruccion en lenguaje ensamblador, puedo trabajar con multiples datos.

Esto nos permite mejorar tiempos de ejecucion (el trabajo lo hago en paralelo y por ende mayor rendimiento), importantes en el procesamiento de imagenes.

Instrucciones

https://www.felixcloutier.com/x86/

▼ Ejemplo: Hello World

```
; Programa helloworld.asm
; Para ensamblar ejecutar:
; nasm -f elf64 helloworld.asm -o helloworld.o
; Para enlazar ejecutar:
; ld helloworld.o -o helloworld
; Para correr el ejecutable:
; ./helloworld

; SEGMENTO DE DATOS
; Se empleara la etiqueta message y se reservaran elementos de 8 bits
; Cada letra de la cadena se corresponde con un elemento de 8 bits
; El numero 10 se corresponde con el caracter \n
section .data
  message db "Hello World",10
  len equ $ - message

; SEGMENTO DE TEXTO
section .text
  global _start

_start:
; LLAMADA AL SISTEMA
; rax => ID <= 1 : sys_write
; rdi => Primer parametro   : output
; rsi => Segundo parametro  : direccion del mensaje
; rdx => Tercer parametro  : longitud del mensaje
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, len
  syscall
; LLAMADA AL SISTEMA
; rax => ID <= 60  : sys_exit
; rdi => Primer parametro   : 0 <= sin errores
  mov rax, 60
  mov rdi, 0
  syscall
```


En la seccion **.data** se ubican las variables y constantes

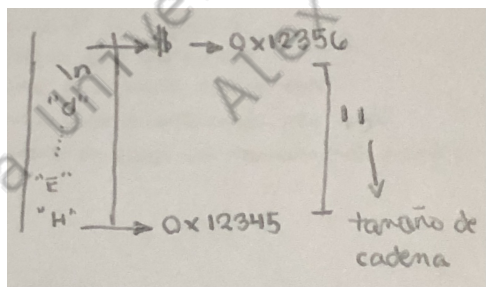
Declaración	Significado
db	Variable de 8 bits
dw	Variable de 16 bits
dd	Variable de 32 bits
dq	Variable de 64 bits

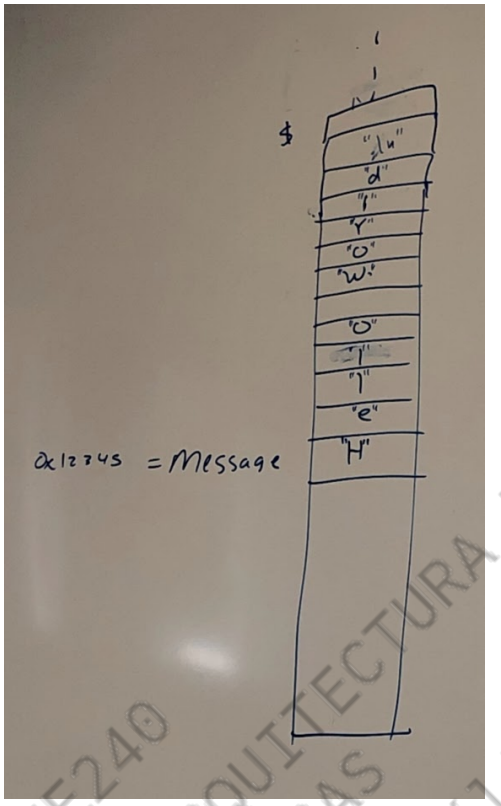
En el siguiente codigo podemos observar lo siguiente:

```
section .data
  message db "Hello World",10
  len equ $ - message
```

message es una etiqueta (posicionada en una direccion de memoria), y a partir de ahi se coloca la frase ***Hello World.*** Tener en consideracion que, el "**,10**" representa \n del salto de linea.

La etiqueta **len** va a representar la diferencia entre la posicion de memoria actual - la posicion de message, de manera que obtenemos el tamanio de la cadena





En la seccion **.text** se encuentra lo siguiente:

```
section .text
  global _start

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, message
  mov rdx, len
  syscall

  mov rax, 60
  mov rdi, 0
  syscall
```

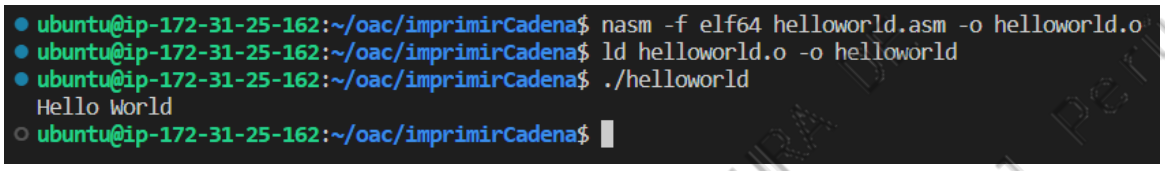
syscall es el llamado al sistema que quieres hacer pero para ello, debemos darle las instrucciones que queremos que este haga. Hay un codigo basado en registros que te permite hacer eso.

```
mov rax, 1 ;indica escritura en el sistema
mov rdi, 1 ;indica salida

;argumentos de entrada
mov rsi, message ;posicion de memoria de message
mov rdx, len ;el tamanio para saber cuantas posiciones a partir de message imprime
```

```
syscall
```

Finalmente, cerramos el programa haciendo un llamado al sistema:

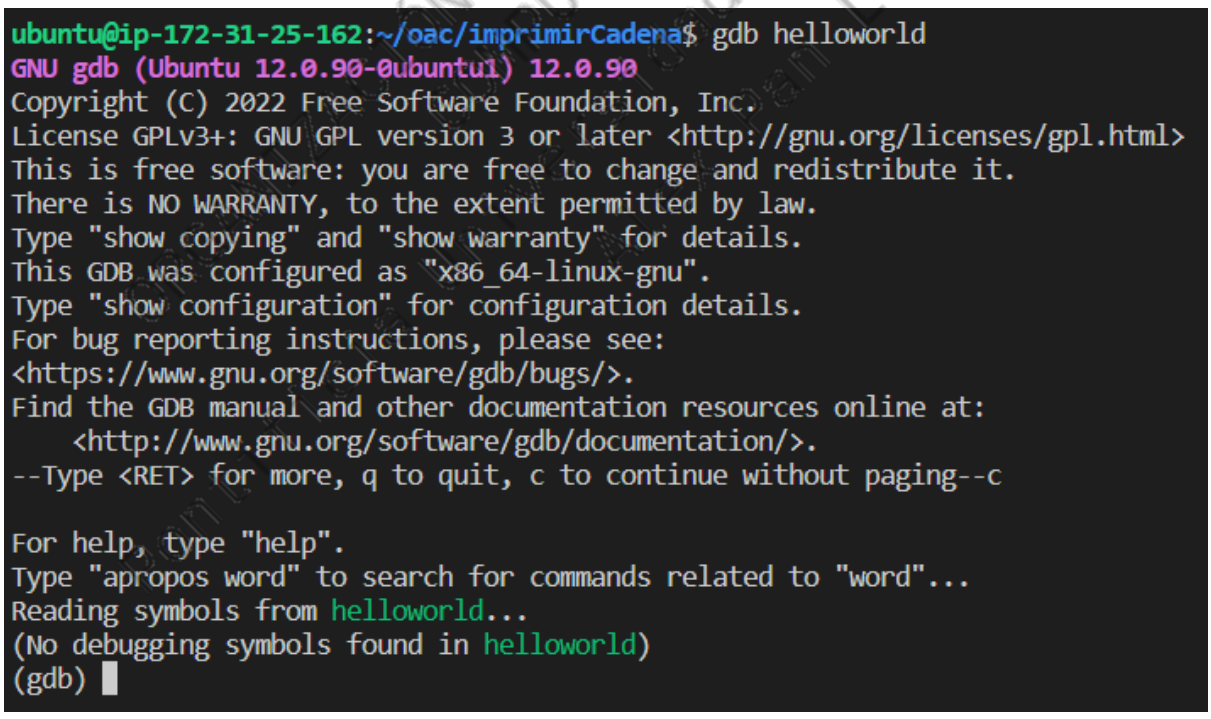
```
mov rax, 60
mov rdi, 0
syscall
```



Uso del GDB

Para correr el depurador:

> 💡 **gdb** [**nombre de archivo ejecutable**]



Para dejar un breakpoint:

💡 **break [parte que quieres revisar]**

Para fijar el tipo de desensamblaje en sintaxis Intel (por defecto esta en sintaxis AT&T):

💡 **set disassembly-flavor intel**

Run para comenzar a depurar de linea en linea segun la posicion que creaste el breakpoint:

💡 **run**

Para desensamblar:

💡 **disass**

Para ir a la siguiente instruccion:

💡 **ni**

Podemos darnos cuenta que lo que esta en < > representa cuanto ha ocupado nuestro programa. En este caso, el primer mov ocupo 5 bytes, el segundo ocupo 5 bytes y el movabs ocupo 10 bytes… asi sucesivamente. Curiosamente, ese 0x402000 era la direccion de memoria de la etiqueta **message**.

```
(gdb) break _start
Breakpoint 1 at 0x401000
(gdb) set disassembly-flavor intel
(gdb) run
Starting program: /home/ubuntu/oac/imprimirCadena/helloworld

Breakpoint 1, 0x0000000000401000 in _start ()
(gdb) disass
Dump of assembler code for function _start:
=> 0x0000000000401000 <+0>:     mov    eax,0x1
   0x0000000000401005 <+5>:     mov    edi,0x1
   0x000000000040100a <+10>:    movabs rsi,0x402000
   0x0000000000401014 <+20>:    mov    edx,0xc
   0x0000000000401019 <+25>:    syscall
   0x000000000040101b <+27>:    mov    eax,0x3c
   0x0000000000401020 <+32>:    mov    edi,0x0
   0x0000000000401025 <+37>:    syscall
End of assembler dump.
(gdb) ni
0x0000000000401005 in _start ()
(gdb) ni
0x000000000040100a in _start ()
(gdb) disass
Dump of assembler code for function _start:
   0x0000000000401000 <+0>:     mov    eax,0x1
   0x0000000000401005 <+5>:     mov    edi,0x1
=> 0x000000000040100a <+10>:    movabs rsi,0x402000
   0x0000000000401014 <+20>:    mov    edx,0xc
   0x0000000000401019 <+25>:    syscall
   0x000000000040101b <+27>:    mov    eax,0x3c
   0x0000000000401020 <+32>:    mov    edi,0x0
   0x0000000000401025 <+37>:    syscall
End of assembler dump.
(gdb)
```

Para poder revisar la informacion de los registros:

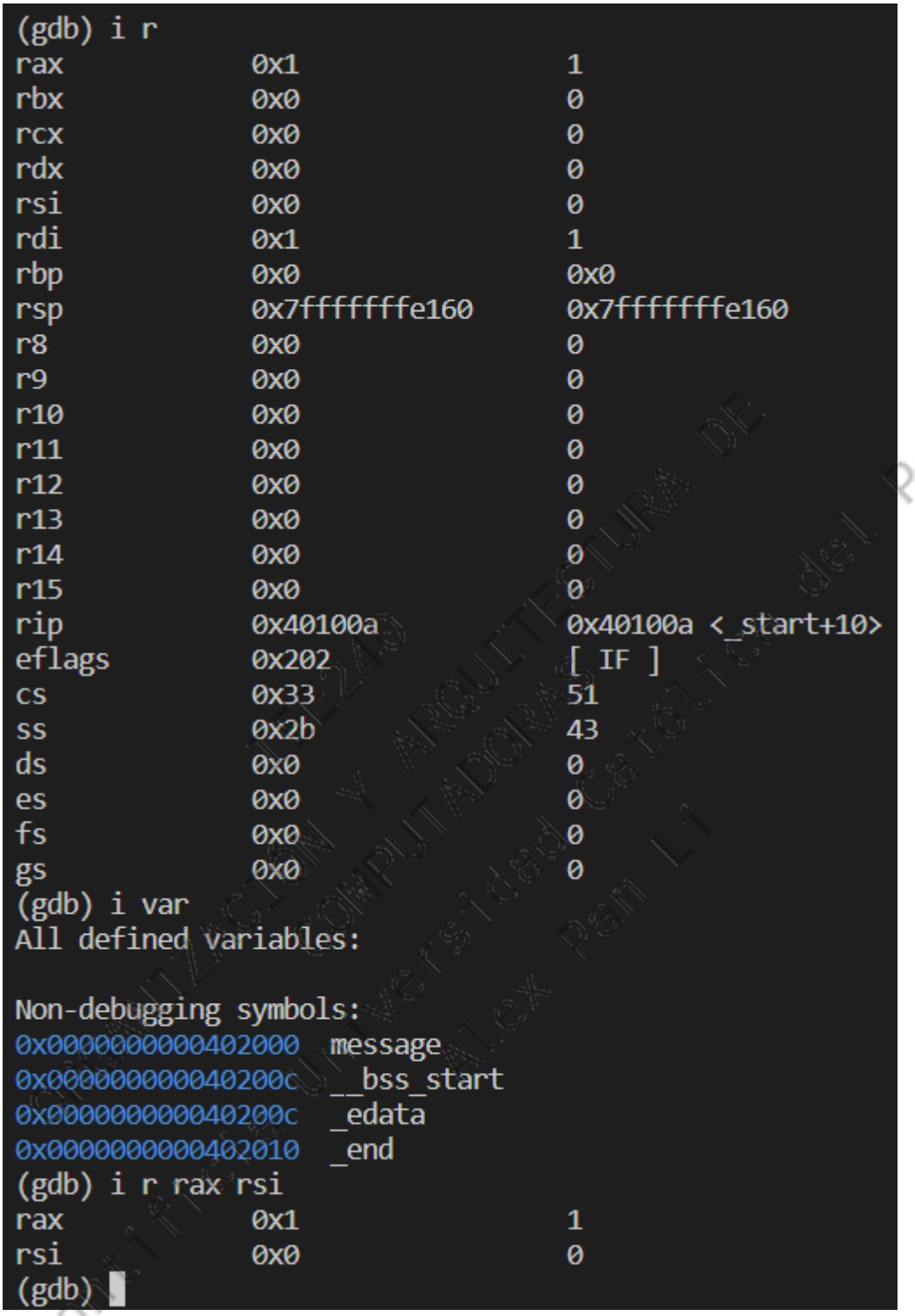
💡 **i r**

Para poder revisar la informacion de las variables:

💡 **i var**

Para poder revisar la informacion de registros especificos:

💡 **i r** [**nombre de registros (si son mas de uno, los agregas sin coma)**]



Para salir de la depuracion:

> 💡 **q**

Para imprimir el contenido de variable:

💡 **print/[formato] (tipo de dato) [nombre de variable]**

Letra	Formato
d	decimal
x	hexadecimal
t	binario
u	sin signo
f	punto flotante
i	instrucción
c	caracter
s	string
a	address

▼ Ejemplo: Get Name

```
; Programa getname.asm
; Para ensamblar ejecutar:
; nasm -f elf64 getname.asm -o getname.o
; Para enlazar ejecutar:
; ld getname.o -o getname
; Para correr el ejecutable:
; ./getname

; SEGMENTO DE DATOS
section .data
  question db "What is your name? "
  lenq equ $ - question
  greet db "Hello, "
  leng equ $ - greet

; SEGMENTO BSS (Block Started by Symbol)
; Reservamos 16 bytes para el nombre que sera ingresado
section .bss
  name resb 16

; SEGMENTO DE TEXTO
```

```
section .text
  global _start

; SYS_WRITE
_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, question
  mov rdx, lenq
  syscall

; SYS_READ
  mov rax, 0
  mov rdi, 0
  mov rsi, name
  mov rdx, 16
  syscall

; SYS_WRITE
  mov rax, 1
  mov rdi, 1
  mov rsi, greet
  mov rdx, leng
  syscall

; SYS_WRITE
  mov rax, 1
  mov rdi, 1
  mov rsi, name
  mov rdx, 16
  syscall

; SYS_EXIT
  mov rax, 60
  mov rdi, 0
  syscall
```


Primero tenemos nuestro segmento **.data** donde encontramos las etiquetas **question** y **greet**, junto con sus respectivos tamanios:

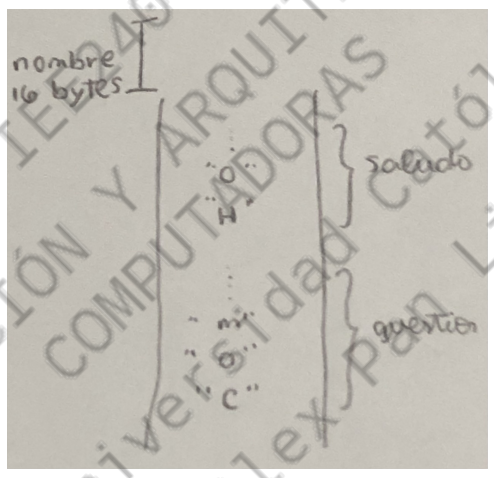
```
section .data
  question db "What is your name? "
  lenq equ $ - question
  greet db "Hello, "
  leng equ $ - greet
```

En la seccion **.bss** reservamos bytes a fin de almacenar ahi nuestro nombre. Tener en consideracion que lo ideal es dejar un espacio prudente de memoria para que ocupe suficientemente cualquier nombre que nosotros insertemos.

Declaración	Significado
resb	Variable de 8 bits
resw	Variable de 16 bits
resd	Variable de 32 bits
resq	Variable de 64 bits

En nuestro caso, reservamos 16 bytes para colocar nuestro nombre

```
section .bss
  name resb 16
```



Imprimimos el “**Como te llamas?** ”

```
mov rax, 1
mov rdi, 1
mov rsi, question
mov rdx, lenq
syscall
```

Luego, el sistema nos espera por una entrada de datos (en este caso por teclado)

```
;realizamos un llamado al sistema de entrada
mov rax, 0
mov rdi, 0
```

```
;se guarda el nombre en el registro
mov rsi, name
;se asigna a la etiqueta name 16 bytes de memoria
mov rdx, 16
syscall
```

Imprimimos el “**Hola,** “

```
mov rax, 1
mov rdi, 1
mov rsi, greet
mov rdx, leng
syscall
```

Imprimimos el nombre

```
mov rax, 1
mov rdi, 1
mov rsi, name
mov rdx, 16
syscall
```

Finalizamos el programa:

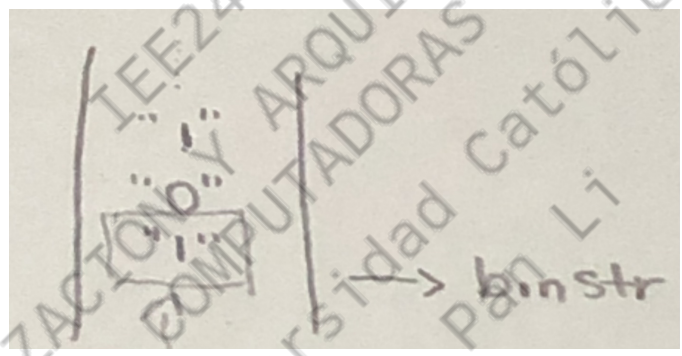
```
mov rax, 60
mov rdi, 0
syscall
```

▼ Ejemplo: Bin to Decimal

```
;Calculo de un número string de 8 bits a decimal almacenado en memoria
section .data
  binstr  db  "10101010"; Número a evaluar
  res db  0   ; Resultado

section .text
  global  _start

_start:
  mov r15,  binstr  ;Guardamos la posición de memoria en R15
  mov r14,  128 ;como son 8 bits, empezamos de 128
  mov r13,  2 ;Divisor de las potencias de 2
  mov r12,  0 ;Acumulador del resultado

mulbuc:
  mov al, [r15] ;Guardamos los 8 bits en AL
```

```
  sub al, '0' ;Restamos los valores ASCII
  mul r14   ;RAX=RAX*R14
  add r12,  rax ;Acumulamos en R12
sigpa:
  xor rdx,  rdx ;Limpiamos RDX (ver instrucciones)
  mov rax,  r14 ;Guardamos en RAX para dividir
  div r13   ;Dividimos entre la potencia de 2
  mov r14,  rax ;Devolvemos  al divisor
  inc r15   ;Incrementamos posición de memoria
  cmp r14,  0 ;CMP con 0 para ver si terminó la cadena
  jne mulbuc

fin:
  mov [res],  r12b
  mov rax,  60
  mov   rdi,  0
  syscall
```

En algun lugar de la memoria RAM estara mi etiqueta **binstr** y la unica diferencia es que es un numero definido en comillas.



Asimismo, la etiqueta **res** es donde guardare el resultado del valor en decimal

```
section .data
  binstr  db  "10101010"; Número a evaluar
  res db  0   ; Resultado
```

```
mov r15,  binstr  ;Guardamos la posición de memoria en R15
mov r14,  128     ;como son 8 bits, empezamos de 128
mov r13,  2       ;Divisor de las potencias de 2
mov r12,  0       ;Acumulador del resultado
```

Primero guardamos el contenido de lo que apunta la direccion. Por lo tanto, **AL = "1"**

```
mov al, [r15] ;Guardamos los 8 bits en AL
```

Restamos el contenido de **AL** por ‘0’ para obtener el valor numerico. Recordar que si tienes un numero ASCII y le restas ‘0’, te da el valor del numero. En este caso si **AL = “1”**, con esta instruccion **AL = 1**

```
sub al, '0' ;Restamos los valores ASCII
```

El resultado de la multiplicacion va a ser un numero sin signo.

```
mul r14   ;RAX=RAX*R14
```

Para entender mejor como funciona la multiplicacion, nos dirigimos al enlace de instrucciones:

Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
MUL *r/m8*	M	Valid	Valid	Unsigned multiply (AX ← AL ∗ *r/m8*).
MUL *r/m8**	M	Valid	N.E.	Unsigned multiply (AX ← AL ∗ *r/m8*).
MUL *r/m16*	M	Valid	Valid	Unsigned multiply (DX:AX ← AX ∗ *r/m16*).
MUL *r/m32*	M	Valid	Valid	Unsigned multiply (EDX:EAX ← EAX ∗ *r/m32*).
MUL *r/m64*	M	Valid	N.E.	Unsigned multiply (RDX:RAX ← RAX ∗ *r/m64*).

Puede ser utilizada por registros o espacios de memoria de 8, 16, 32 o 64 bits. Por ejemplo para un registro de 8, el resultado es el producto de un registro (el argumento de entrada que insertamos) con AL y este resultado se guarda en AX.

En nuestro caso, que son 64 bits, la parte menos significativa se chanca en RAX y la parte mas significativa se guarda en RDX

Recordemos que previamente **AX = 1**, entonces **EAX = 1** y por ende **RAX = 1**. Por otro lado, **R14 = 128**

$$RAX = RAX * R14 = 1 * 128$$

Acumulamos el resultado de lo que quedo de RAX en R12

```
add r12,  rax ;Acumulamos en R12
```

Limpiamos RDX (luego veremos por que)

```
xor rdx,  rdx ;Limpiamos RDX (ver instrucciones)
```

Movemos lo que hay en R14 en RAX. Ahora **RAX = 128**

```
mov rax,  r14 ;Guardamos en RAX para dividir
```

Utilizamos la instruccion de division entre lo que hay en RAX con el registro R13 = 2.

```
div r13   ;Dividimos entre la potencia de 2
```

Guardamos el cociente en RAX y el residuo en RDX. Por eso es que habiamos limpiado RDX, pues si no lo haciamos, dependiendo del tamanio, podria tener cualquier valor a pesar que la instruccion mul lo chancaba (es un por si acaso).

Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
DIV *r/m8*	M	Valid	Valid	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
DIV *r/m8**	M	Valid	N.E.	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
DIV *r/m16*	M	Valid	Valid	Unsigned divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
DIV *r/m32*	M	Valid	Valid	Unsigned divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.
DIV *r/m64*	M	Valid	N.E.	Unsigned divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder.

Por ser la primera iteracion, despues de esta instruccion, **RAX = 128** ahora sera **RAX = 64**

$$RAX = RAX/2 = 128/2 = 64$$

El cociente lo guardamos en R14

```
mov r14,  rax ;Devolvemos  al divisor
```

Incrementamos la posicion de memoria de tal forma que si en la primera iteracion de nuestro **binstr** "**10101010**" estabamos en el primer 1, ahora nos movemos a la siguiente posicion que es 0

```
inc r15   ;Incrementamos posición de memoria
```

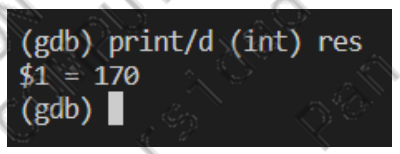
Comparamos si R14 es 0 (eso nos dice que ya acabo la cadena). En caso sea 0, el jump **JNE** no se activa y nos pasamos al fin del programa, caso contrario, regresamos a mulbuc para volver a acumular

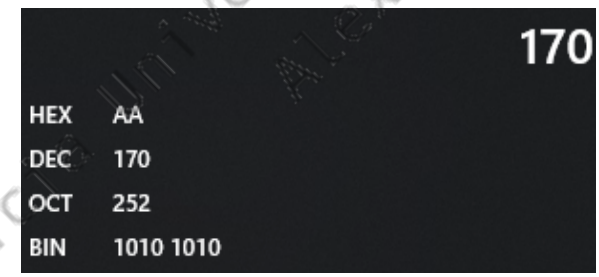
```
cmp r14,  0 ;CMP con 0 para ver si terminó la cadena
jne mulbuc
```

Fin del programa

```
fin:
  ;la parte low de R12 estara la respuesta y se lo manda a la direccion de res
  mov [res],  r12b
  mov rax,  60
  mov   rdi,  0
  syscall
```

Verificamos si tenemos el valor decimal del string entregado…





▼ Ejemplo: Invertir Cadena

```
section .data
  mensaje db "Hola Mundo",0

section .bss
  inverse resb 20

section .text
  global _start
```

```
_start:

  mov rax, mensaje
  mov rbx,0

_countLoop:

  inc rax
  inc rbx
  mov cl, [rax]
  cmp cl,0
  jne _countLoop

  mov r8,rbx
  mov r9, inverse
  mov rax,mensaje
  add rax,r8

_reverse:

  dec rax
  mov r10,[rax]
  mov [r9],r10
  inc r9
  cmp rax, mensaje
  jne _reverse

  mov [r9], byte 10
  inc rbx

  mov rax,1
  mov rdi,1
  mov rsi, inverse
  mov rdx,rbx
  syscall

  mov rax,60
  mov rdi,0
  syscall
```


Nuestra etiqueta **mensaje** representa el string "***Hola Mundo***" con un "***,0***" que representa fin de la cadena

```
section .data
  mensaje db "Hola Mundo",0
```

Por otro lado, especificamos el espacio que se le designa a la cadena. En este caso, reservamos 20 bytes

```
section .bss
  inverse resb 20
```

Almaceno la direccion de memoria de mensaje al registro RAX y almaceno 0 en RBX

```
mov rax, mensaje
mov rbx, 0
```

Incrementamos **RAX = RAX + 1** y **RBX = RBX + 1**

```
_countLoop:
  inc rax
  inc rbx
```

Movemos lo que era en RAX a CL (este es un registro de 8 bits, y es el primitivo del registro RCX de 64 bits que conocemos)

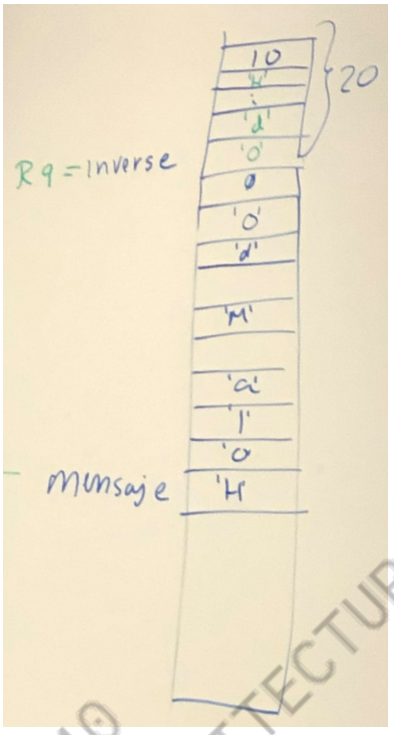
```
mov cl, [rax]
```

Comparo si CL es 0 y si no es igual, me quedo en loop hasta que CL sea 0. Lo que estoy haciendo ahi es contar el numero de letras (es lo mismo con lo que hemos hecho con las etiquetas len para obtener el tamanio de la etiqueta). Tener en consideracion que el contenido de RAX es "***Hola Mundo***",0 por lo tanto, en algun momento ese CL se compara con ese 0 que estaba al ultimo de RAX y ahi es cuando acaba el loop.

```
cmp cl,0
jne _countLoop
```

Una vez se cuente el numero de letras, RBX ya habra tenido el numero de letras. La direccion de memoria de RBX (que representa el numero de letras) e **inverse** se guarda en R8 y R9 (este registro apuntara la direccion por donde comenzara el string invertido), mientras que la direccion de memoria de **mensaje** se chanca en RAX

```
mov r8,rbx
mov r9, inverse
mov rax,mensaje
```



Sumo la posicion de memoria de **mensaje** con el tamanio que estaba en el registro R8 para que RAX se encuentre en la posicion del contador

```
add rax,r8
```

Decrementamos RAX de tal forma que ahora nos ubiquemos en “o”

“Hola Mundo”,0

```
_reverse:
  dec rax
```

El contenido de RAX (en este caso “o”) estara almacenada en R10

```
mov r10,[rax]
```

Almacenamos el valor de R10 (que en este caso es “o”) dentro de R9 (registro que apuntaba a **inverse**). Esto es basicamente decir que **inverse**[0] = ‘o’

```
mov [r9],r10
```

Incrementamos la posicion de R9, de tal forma que pongamos en esa nueva posicion la siguiente letra (que es “d”). Esto es como decir R9 = **inverse** + 1

```
inc r9
```

Se compara RAX con la etiqueta **mensaje**. Como estoy decrementando RAX, luego comparo si RAX llego hasta la posicion de esa etiqueta (que marca el inicio del string). En caso haya llegado, paso al siguiente instruccion, de lo contrario, regreso al loop.

```
cmp rax, mensaje
jne _reverse
```

Una vez que obtengamos el string invertido, el valor de R9 que sera la ultima letra del string invertido + 1, tendra el valor de 10, que representa nuestro cambio de linea \n

“***odnuM aloH***”,10

```
mov [r9], byte 10
```

Incremento RBX para que salga el salto de linea

```
inc rbx
```

Imprimo el string invertido

```
mov rax,1
mov rdi,1
mov rsi, inverse
mov rdx,rbx
syscall
```

Finalizo el programa

```
mov rax,60
mov rdi,0
syscall
```